Laboratorio di Elementi di Bioinformatica

Laurea Triennale in Informatica
(codice: E3101Q116)

AA 2016/2017

Linguaggio Ruby: blocchi e iteratori

Docente: Raffaella Rizzi

Blocco

Un blocco è una porzione di codice racchiusa tra:

- parentesi graffe (se il blocco è su un'unica riga)
- le parole chiave `do` ed `end` (se il blocco è su più righe)

Blocco

Un blocco è una porzione di codice racchiusa tra:

- parentesi graffe (se il blocco è su un'unica riga)
- le parole chiave `do` ed `end` (se il blocco è su più righe)

In Ruby un blocco è da intendere come il corpo di un metodo senza nome che (al pari di un metodo) può ricevere parametri e restituire un valore

Blocco

Un blocco è una porzione di codice racchiusa tra:

- parentesi graffe (se il blocco è su un'unica riga)
- le parole chiave `do` ed `end` (se il blocco è su più righe)

In Ruby un blocco è da intendere come il corpo di un metodo senza nome che (al pari di un metodo) può ricevere parametri e restituire un valore

Blocco su una sola riga

```
{|par1, par2, …, parN| block_on_one_row}
```

Blocco

Un blocco è una porzione di codice racchiusa tra:

- parentesi graffe (se il blocco è su un'unica riga)
- le parole chiave `do` ed `end` (se il blocco è su più righe)

In Ruby un blocco è da intendere come il corpo di un metodo senza nome che (al pari di un metodo) può ricevere parametri e restituire un valore

Blocco su una sola riga

```
{|par1, par2, …, parN| block_on_one_row}
```

Blocco su più righe

```
do |par1, par2, …, parN|

 block_on_many_rows

end
```

Blocco

Un blocco è una porzione di codice racchiusa tra:

- parentesi graffe (se il blocco è su un'unica riga)
- le parole chiave `do` ed `end` (se il blocco è su più righe)

In Ruby un blocco è da intendere come il corpo di un metodo senza nome che (al pari di un metodo) può ricevere parametri e restituire un valore

Blocco su una sola riga

```
{|string1, string2| puts string1; puts string2; string1+string2}
```

Blocco su più righe

```
do |par1, par2, …, parN|

 block_on_many_rows

end
```

Blocco

Un blocco è una porzione di codice racchiusa tra:

- parentesi graffe (se il blocco è su un'unica riga)
- le parole chiave `do` ed `end` (se il blocco è su più righe)

In Ruby un blocco è da intendere come il corpo di un metodo senza nome che (al pari di un metodo) può ricevere parametri e restituire un valore

Blocco su una sola riga

```
{|string1, string2| puts string1; puts string2; string1+string2}
```

Lo stesso blocco su più righe

```
do |string1, string2|
 puts string1
 puts string2
 string1+string2
end
```

Blocco

Un blocco è una porzione di codice racchiusa tra:

- parentesi graffe (se il blocco è su un'unica riga)
- le parole chiave `do` ed `end` (se il blocco è su più righe)

In Ruby un blocco è da intendere come il corpo di un metodo senza nome che (al pari di un metodo) può ricevere parametri e restituire un valore

Blocco su una sola riga

```
{|string1, string2| puts string1; puts string2; string1+string2}
```

Lo stesso blocco su più righe

```
do |string1, string2|
 puts string1
 puts string2
 string1+string2
end
```

`string1` e `string2` sono i due parametri del blocco (da racchiudere tra barre verticali)

Blocco

Un blocco è una porzione di codice racchiusa tra:

- parentesi graffe (se il blocco è su un'unica riga)
- le parole chiave `do` ed `end` (se il blocco è su più righe)

In Ruby un blocco è da intendere come il corpo di un metodo senza nome che (al pari di un metodo) può ricevere parametri e restituire un valore

Blocco su una sola riga

```
{|string1, string2| puts string1; puts string2; string1+string2}
```

Lo stesso blocco su più righe

```
do |string1, string2|
 puts string1
 puts string2
 string1+string2
end
```

La concatenazione di `string1` e di `string2` è il valore restituito

Blocco

Un blocco è una porzione di codice racchiusa tra:

- parentesi graffe (se il blocco è su un'unica riga)
- le parole chiave `do` ed `end` (se il blocco è su più righe)

In Ruby un blocco è da intendere come il corpo di un metodo senza nome che (al pari di un metodo) può ricevere parametri e restituire un valore

Blocco su una sola riga

```
{|string1, string2| puts string1; puts string2; string1+string2}
```

Lo stesso blocco su più righe

```
do |string1, string2|
 puts string1
 puts string2
 string1+string2
end
```

Attenzione! Un blocco da solo non funziona

Per fare funzionare un blocco…

- Il blocco deve sempre essere invocato da un metodo e pertanto deve apparire accanto all'invocazione di un metodo

Per fare funzionare un blocco…

- Il blocco deve sempre essere invocato da un metodo e pertanto deve apparire accanto all'invocazione di un metodo
- Il metodo al suo interno dovrà provvedere a invocare il blocco (passandogli eventualmente i parametri) attraverso la parola chiave `yield`

Per fare funzionare un blocco…

- Il blocco deve sempre essere invocato da un metodo e pertanto deve apparire accanto all'invocazione di un metodo
- Il metodo al suo interno dovrà provvedere a invocare il blocco (passandogli eventualmente i parametri) attraverso la parola chiave `yield`

```
def invoca_due_volte

  yield
  yield

end

invoca_due_volte {puts “Hello”}
```

Per fare funzionare un blocco…

- Il blocco deve sempre essere invocato da un metodo e pertanto deve apparire accanto all'invocazione di un metodo
- Il metodo al suo interno dovrà provvedere a invocare il blocco (passandogli eventualmente i parametri) attraverso la parola chiave `yield`

```
def invoca_due_volte

  yield
  yield

end

invoca_due_volte {puts "Hello"}
```

Blocco (senza parametri) composto dall'unica istruzione `puts "Hello"`

Per fare funzionare un blocco…

- Il blocco deve sempre essere invocato da un metodo e pertanto deve apparire accanto all'invocazione di un metodo
- Il metodo al suo interno dovrà provvedere a invocare il blocco (passandogli eventualmente i parametri) attraverso la parola chiave `yield`

```
def invoca_due_volte

  yield
  yield

end

invoca_due_volte {puts "Hello"}
```

Il metodo `invoca_due_volte` è il metodo invocante…

Per fare funzionare un blocco…

- Il blocco deve sempre essere invocato da un metodo e pertanto deve apparire accanto all'invocazione di un metodo
- Il metodo al suo interno dovrà provvedere a invocare il blocco (passandogli eventualmente i parametri) attraverso la parola chiave `yield`

```
def invoca_due_volte

  yield
  yield

end

invoca_due_volte {puts "Hello"}
```

… che invoca due volte il blocco che gli sta accanto

Per fare funzionare un blocco…

- Il blocco deve sempre essere invocato da un metodo e pertanto deve apparire accanto all'invocazione di un metodo
- Il metodo al suo interno dovrà provvedere a invocare il blocco (passandogli eventualmente i parametri) attraverso la parola chiave `yield`

```
def invoca_due_volte

  yield
  yield

end

inv
```

```
>Hello
>Hello
>
```

Per fare funzionare un blocco…

- Il blocco deve sempre essere invocato da un metodo e pertanto deve apparire accanto all'invocazione di un metodo
- Il metodo al suo interno dovrà provvedere a invocare il blocco (passandogli eventualmente i parametri) attraverso la parola chiave `yield`

```
def invoca_due_volte

  yield
  yield

end

invoca_due_volte {a=1; b=2; puts a+b}
```

Cambiamo il blocco!

Per fare funzionare un blocco…

- Il blocco deve sempre essere invocato da un metodo e pertanto deve apparire accanto all'invocazione di un metodo
- Il metodo al suo interno dovrà provvedere a invocare il blocco (passandogli eventualmente i parametri) attraverso la parola chiave `yield`

```
def invoca_due_volte

  yield
  yield

end

inv
```

```
>3
>3
>
```

Parametri di un blocco

Un blocco può ricevere parametri dal metodo invocante

Parametri di un blocco

Un blocco può ricevere parametri dal metodo invocante

```
def calcola_somma(n1, n2)

  somma = n1+n2
  yield(somma)

end

calcola_somma(2, 10) {|s| puts s}
```

Parametri di un blocco

Un blocco può ricevere parametri dal metodo invocante

```
def calcola_somma(n1, n2)

  somma = n1+n2
  yield(somma)

end

calcola_somma(2, 10) {|s| puts s}
```

Blocco che riceve il parametro `s`

Parametri di un blocco

Un blocco può ricevere parametri dal metodo invocante

```
def calcola_somma(n1, n2)

  somma = n1+n2
  yield(somma)

end

calcola_somma(2, 10) {|s| puts s}
```

Metodo invocante `calcola_somma`

Parametri di un blocco

Un blocco può ricevere parametri dal metodo invocante

```
def calcola_somma(n1, n2)

  somma = n1+n2
  yield(somma)

end

calcola_somma(2, 10) {|s| puts s}
```

Il metodo invocante `calcola_somma` invoca il blocco passandogli come paramtero la variabile `somma`

Parametri di un blocco

Un blocco può ricevere parametri dal metodo invocante

```
def calcola_somma(n1, n2)

  somma = n1+n2
  yield(somma)

end

calcola_somma(2, 10) {|s| puts s}
```

Il metodo invocante `calcola_somma` invoca il blocco passandogli come paramtero la variabile `somma`

```
>12
>
```

Parametri di un blocco

Un blocco può ricevere parametri dal metodo invocante

```
def calcola_somma(n1, n2)

  somma = n1+n2
  yield(somma)

end

calcola_somma(2, 10) do|s|
                       s=s*10
                       puts s
                     end
```

Cambiamo il blocco! Da notare che è stato scritto su più righe

Parametri di un blocco

Un blocco può ricevere parametri dal metodo invocante

```
def calcola_somma(n1, n2)

  somma = n1+n2
  yield(somma)

end

calcola_somma(2, 10) do|s|
                       s=s*10
                       puts s
                     end
```

Cambiamo il blocco! Da notare che è stato scritto su più righe

```
>120
>
```

Valore restituito da un blocco

Un blocco restituisce un valore che il metodo invocante può utilizzare

```
def ottieni_qualcosa(n1, n2)

  qualcosa = yield(n1, n2)
  puts qualcosa

end

ottieni_qualcosa(2, 10) {|b1, b2| b1+b2}
```

Valore restituito da un blocco

Un blocco restituisce un valore che il metodo invocante può utilizzare

```
def ottieni_qualcosa(n1, n2)

  qualcosa = yield(n1, n2)
  puts qualcosa

end

ottieni_qualcosa(2, 10) {|b1, b2| b1+b2}
```

Il blocco restituisce la somma dei suoi due parametri `b1` e `b2`

Valore restituito da un blocco

Un blocco restituisce un valore che il metodo invocante può utilizzare

```
def ottieni_qualcosa(n1, n2)

  qualcosa = yield(n1, n2)
  puts qualcosa

end

ottieni_qualcosa(2, 10) {|b1, b2| b1+b2}
```

Il metodo invoca il blocco e assegna il valore restituito alla variabile `qualcosa`

Valore restituito da un blocco

Un blocco restituisce un valore che il metodo invocante può utilizzare

```
def ottieni_qualcosa(n1, n2)

  qualcosa = yield(n1, n2)
  puts qualcosa

end

ottieni_qualcosa(2, 10) {|b1, b2| b1+b2}
```

Il metodo invoca il blocco e assegna il valore restituito alla variabile `qualcosa`

```
>12
>
```

Valore restituito da un blocco

Un blocco restituisce un valore che il metodo invocante può utilizzare

```
def ottieni_qualcosa(n1, n2)

  qualcosa = yield(n1, n2)
  puts qualcosa

end

ottieni_qualcosa(2, 10) {|b1, b2| b1*b2}
```

Cambiamo il blocco!

Valore restituito da un blocco

Un blocco restituisce un valore che il metodo invocante può utilizzare

```
def ottieni_qualcosa(n1, n2)

  qualcosa = yield(n1, n2)
  puts qualcosa

end

ottieni_qualcosa(2, 10) {|b1, b2| b1*b2}
```

Cambiamo il blocco!

```
>20
>
```

Controllo dell'esistenza di un blocco

Il metodo invocante può controllare se gli associato un blocco tramite la funzione `block_given?`

Controllo dell'esistenza di un blocco

Il metodo invocante può controllare se gli associato un blocco tramite la funzione `block_given?`

```
def ottieni_qualcosa(n1, n2)

  if block_given?
    qualcosa = yield(n1, n2)
  else
    qualcosa = "niente"
  end
  puts qualcosa

end

ottieni_qualcosa(2, 10) {|b1, b2| b1+b2}
ottieni_qualcosa(2, 10)
```

Controllo dell'esistenza di un blocco

Il metodo invocante può controllare se gli associato un blocco tramite la funzione `block_given?`

```
def ottieni_qualcosa(n1, n2)

  if block_given?
    qualcosa = yield(n1, n2)
  else
    qualcosa = "niente"
  end
  puts qualcosa

end

ottieni_qualcosa(2, 10) {|b1, b2| b1+b2}
ottieni_qualcosa(2, 10)
```

Solo se `block_given?` restituisce `true` viene invocato il blocco

Controllo dell'esistenza di un blocco

Il metodo invocante può controllare se gli associato un blocco tramite la funzione `block_given?`

```
def ottieni_qualcosa(n1, n2)

  if block_given?
    qualcosa = yield(n1, n2)
  else
    qualcosa = "niente"
  end
  puts qualcosa

end

ottieni_qualcosa(2, 10) {|b1, b2| b1+b2}
ottieni_qualcosa(2, 10)
```

Alla seconda invocazione di `ottieni_qualcosa` non è associato nessun blocco

Controllo dell'esistenza di un blocco

Il metodo invocante può controllare se gli associato un blocco tramite la funzione `block_given?`

```
def ottieni_qualcosa(n1, n2)

  if block_given?
    qualcosa = yield(n1, n2)
  else
    qualcosa = "niente"
  end
  puts qualcosa

end

ottieni_qualcosa(2, 10) {|b1, b2| b1+b2}
ottieni_qualcosa
```

```
>12
>niente
>
```

Gestione delle variabili di un blocco

REGOLA 1: se esiste una variabile nel blocco che ha lo stesso nome di una variabile esterna al blocco, allora si ha a che fare con la stessa variabile.

Gestione delle variabili di un blocco

REGOLA 1: se esiste una variabile nel blocco che ha lo stesso nome di una variabile esterna al blocco, allora si ha a che fare con la stessa variabile.

```
def ottieni_qualcosa(n1, n2)

  if block_given?
    qualcosa = yield(n1, n2)
  else
    qualcosa = “niente”
  end
  puts qualcosa

end

s = 100
ottieni_qualcosa(2, 10) {|b1, b2| s+b1+b2}
```

Gestione delle variabili di un blocco

REGOLA 1: se esiste una variabile nel blocco che ha lo stesso nome di una variabile esterna al blocco, allora si ha a che fare con la stessa variabile.

```
def ottieni_qualcosa(n1, n2)

  if block_given?
    qualcosa = yield(n1, n2)
  else
    qualcosa = “niente”
  end
  puts qualcosa

end

s = 100
ottieni_qualcosa(2, 10) {|b1, b2| s+b1+b2}
```

Esiste una variabile `s` che è esterna al blocco

Variabili di un blocco

REGOLA 1: se esiste una variabile nel blocco che ha lo stesso nome di una variabile esterna al blocco, allora si ha a che fare con la stessa variabile.

```
def ottieni_qualcosa(n1, n2)

  if block_given?
    qualcosa = yield(n1, n2)
  else
    qualcosa = “niente”
  end
  puts qualcosa

end

s = 100
ottieni_qualcosa(2, 10) {|b1, b2| s+b1+b2}
```

Esiste una variabile `s` che è interna al blocco

Variabili di un blocco

REGOLA 1: se esiste una variabile nel blocco che ha lo stesso nome di una variabile esterna al blocco, allora si ha a che fare con la stessa variabile.

```
def ottieni_qualcosa(n1, n2)

  if block_given?
    qualcosa = yield(n1, n2)
  else
    qualcosa = “niente”
  end
  puts qualcosa

end

s = 100
ottieni_qualcosa(2, 10) {|b1, b2| s+b1+b2}
```

Sono la stessa variabile!

Variabili di un blocco

REGOLA 1: se esiste una variabile nel blocco che ha lo stesso nome di una variabile esterna al blocco, allora si ha a che fare con la stessa variabile.

```
def ottieni_qualcosa(n1, n2)

  if block_given?
    qualcosa = yield(n1, n2)
  else
    qualcosa = “niente”
  end
  puts qualcosa

end

s =
ott
```

Sono la stessa variabile!

```
>112
>
```

Variabili di un blocco

Per fare in modo che una variabile interna a un blocco, che compare anche all'esterno, sia locale al blocco occorre elencarla dopo i parametri separandola tramite punto e virgola.

Variabili di un blocco

Per fare in modo che una variabile interna a un blocco, che compare anche all'esterno, sia locale al blocco occorre elencarla dopo i parametri separandola tramite punto e virgola.

```
def ottieni_qualcosa(n1, n2)

  if block_given?
    qualcosa = yield(n1, n2)
  else
    qualcosa = "niente"
  end
  puts qualcosa

end

s = 100
ottieni_qualcosa(2, 10) {|b1, b2; s| s=0; s+b1+b2}
puts "Variabile esterna = #{s}"
```

Variabili di un blocco

Per fare in modo che una variabile interna a un blocco, che compare anche all'esterno, sia locale al blocco occorre elencarla dopo i parametri separandola tramite punto e virgola.

```
def ottieni_qualcosa(n1, n2)

  if block_given?
    qualcosa = yield(n1, n2)
  else
    qualcosa = "niente"
  end
  puts qualcosa

end

s = 100
ottieni_qualcosa(2, 10) {|b1, b2; s| s=0; s+b1+b2}
puts "Variabile esterna = #{s}"
```

Variabile locale `s` interna al blocco

Variabili di un blocco

Per fare in modo che una variabile interna a un blocco, che compare anche all'esterno, sia locale al blocco occorre elencarla dopo i parametri separandola tramite punto e virgola.

```
def ottieni_qualcosa(n1, n2)

  if block_given?
    qualcosa = yield(n1, n2)
  else
    qualcosa = "niente"
  end
  puts qualcosa

end

s = 100
ottieni_qualcosa(2, 10) {|b1, b2; s| s=0; s+b1+b2}
puts "Variabile esterna = #{s}"
```

Variabile `s` esterna al blocco

Variabili di un blocco

Per fare in modo che una variabile interna a un blocco, che compare anche all'esterno, sia locale al blocco occorre elencarla dopo i parametri separandola tramite punto e virgola.

```
def ottieni_qualcosa(n1, n2)

  if block_given?
    qualcosa = yield(n1, n2)
  else
    qualcosa = "niente"
  end
  puts qualcosa

end

s =
ott
puts "variabile esterna = #{s}"
```

```
>12
>Variabile esterna = 100
>
```

Variabili di un blocco

REGOLA 2: gli argomenti di un blocco sono sempre locali al blocco anche se all'esterno esistono variabili con lo stesso nome.

Variabili di un blocco

REGOLA 2: gli argomenti di un blocco sono sempre locali al blocco anche se all'esterno esistono variabili con lo stesso nome.

```
def ottieni_qualcosa(n1, n2)

  if block_given?
    qualcosa = yield(n1, n2)
  else
    qualcosa = "niente"
  end
  puts qualcosa

end

b1 = 0; b2 = 0
ottieni_qualcosa(2, 10) {|b1, b2| b1+b2}
```

Variabili di un blocco

REGOLA 2: gli argomenti di un blocco sono sempre locali al blocco anche se all'esterno esistono variabili con lo stesso nome.

```
def ottieni_qualcosa(n1, n2)

  if block_given?
    qualcosa = yield(n1, n2)
  else
    qualcosa = "niente"
  end
  puts qualcosa

end

b1 = 0; b2 = 0
ottieni_qualcosa(2, 10) {|b1, b2| b1+b2}
```

I parametri `b1` e `b2` sono locali al blocco

Variabili di un blocco

REGOLA 2: gli argomenti di un blocco sono sempre locali al blocco anche se all'esterno esistono variabili con lo stesso nome.

```
def ottieni_qualcosa(n1, n2)

  if block_given?
    qualcosa = yield(n1, n2)
  else
    qualcosa = "niente"
  end
  puts qualcosa

end

b1 = 0; b2 = 0
ottieni_qualcosa(2, 10) {|b1, b2| b1+b2}
```

Le due variabili esterne al blocco `b1` e `b2` non sono i parametri del blocco

Variabili di un blocco

REGOLA 2: gli argomenti di un blocco sono sempre locali al blocco anche se all'esterno esistono variabili con lo stesso nome.

```
def ottieni_qualcosa(n1, n2)

  if block_given?
    qualcosa = yield(n1, n2)
  else
    qualcosa = "niente"
  end
  puts qualcosa

end
```

Le due variabili esterne al blocco `b1` e `b2` non sono i parametri del blocco

```
>12
>
```

Cos’è un iteratore?

Un iteratore è un metodo che invoca ripetutamente un blocco

Cos'è un iteratore?

Un iteratore è un metodo che invoca ripetutamente un blocco

```
def iterator_example(array)

  i = 0
  while i < array.length
    yield array[i]
    i = i+1
  end

end

array = [2,4,6,8,10]
iterator_example(array) block???
```

Cos'è un iteratore?

Un iteratore è un metodo che invoca ripetutamente un blocco

```
def iterator_example(array)

  i = 0
  while i < array.length
    yield array[i]
    i = i+1
  end

end

array = [2,4,6,8,10]
iterator_example(array)
```

Il ciclo di iterazione `while` in Ruby ha la seguente sintassi:

```
while condition

  instructions

end
```

dove *`condition`* è un'espressione che restituisce `true` oppure `false`

Cos'è un iteratore?

Un iteratore è un metodo che invoca ripetutamente un blocco

```
def iterator_example(array)

  i = 0
  while i < array.length
    yield array[i]
    i = i+1
  end

end

array = [2,4,6,8,10]
iterator_example(array) block???
```

Il metodo `iterator_example` invoca un blocco per ognuno degli elementi contenuti in `array`, passandoli al blocco come parametro, Al blocco è demandato il compito di manipolarli.

Cos'è un iteratore?

Un iteratore è un metodo che invoca ripetutamente un blocco

```
def iterator_example(array)

  i = 0
  while i < array.length
    yield array[i]
    i = i+1
  end

end

array = [2,4,6,8,10]
iterator_example(array) {|v| puts v}
```

Ad esempio si può stampare l'elemento passato come parametro

Cos'è un iteratore?

Un iteratore è un metodo che invoca ripetutamente un blocco

```
def iterator_example(array)

  i = 0
  while i < array.length
    yield array[i]
    i = i+1
  end

end

array = [2,4,6,8,10]
iterator_example(array) {|v| v=v+1; puts v}
```

…oppure stamparlo dopo averlo incrementato di 1…

Cos'è un iteratore?

Un iteratore è un metodo che invoca ripetutamente un blocco

```
def iterator_example(array)

  i = 0
  while i < array.length
    yield array[i]
    i = i+1
  end

end

array = [2,4,6,8,10]
iterator_example(array) {|v| puts v>10}
```

…oppure testare se è maggiore di 10

Iteratori della classe `Array`

```
array_ref.each
```

L'iteratore `each` invoca un blocco per ciascuno degli elementi dell'array invocante, passandogli tale elemento come parametro

Iteratori della classe `Array`

```
array_ref.each
```

L'iteratore `each` invoca un blocco per ciascuno degli elementi dell'array invocante, passandogli tale elemento come parametro

```
array = [2,4,6,8,10]

array.each {|v| puts v}
```

Iteratori della classe `Array`

```
array_ref.each
```

L'iteratore `each` invoca un blocco per ciascuno degli elementi dell'array invocante, passandogli tale elemento come parametro

```
array = [2,4,6,8,10]

array.each {|v| puts v}
```

```
>2
>4
>6
>8
>10
>
```

Iteratori della classe `Array`

```
array_ref.each
```

L'iteratore `each` invoca un blocco per ciascuno degli elementi dell'array invocante, passandogli tale elemento come parametro

```
array = [2,4,6,8,10]

array.each {|v| puts v
```

```
>2
>4
>6
>8
>10
>
```

Ciò che fa l'iteratore `each` della classe `Array` equivale a ciò che faceva il metodo `iterator_example` dell'esercizio precedente

Iteratori della classe `Array`

```
array_ref.each
```

L'iteratore `each` invoca un blocco per ciascuno degli elementi dell'array invocante, passandogli tale elemento come parametro

```
array = [[3,6,9],[12,15], false]

array.each {|v| puts v}
```

Iteratori della classe `Array`

```
array_ref.each
```

L'iteratore `each` invoca un blocco per ciascuno degli elementi dell'array invocante, passandogli tale elemento come parametro

```
array = [[3,6,9],[12,15], false]

array.each {|v| puts v}
```

```
>3
>6
>9
>12
>15
>false
>
```

Iteratori della classe `Array`

```
array_ref.each
```

L'iteratore `each` invoca un blocco per ciascuno degli elementi dell'array invocante, passandogli tale elemento come parametro

```
array = [[3,6,9],[12,15], false]

array.each {|v| puts v}
```

```
>3
>6
>9
>12
>15
>false
>
```

L'iteratore `each` itera anche su eventuali array annidati

Iteratori della classe `Array`

```
array_ref.find
```

L'iteratore `find` invoca un blocco per ciascun elemento dell'array invocante, passando tale elemento come parametro. Non appena il blocco restituisce un valore diverso da `false` o da `nil`, l'iteratore restituisce il valore dell'elemento passato come parametro

Iteratori della classe `Array`

```
array_ref.find
```

L'iteratore `find` invoca un blocco per ciascun elemento dell'array invocante, passando tale elemento come parametro. Non appena il blocco restituisce un valore diverso da `false` o da `nil`, l'iteratore restituisce il valore dell'elemento passato come parametro

```
array = [3,6,9,12,15]

value = array.find {|v| v > 9}
puts "First value greater than 9: =" + value.to_s
```

Iteratori della classe `Array`

```
array_ref.find
```

L'iteratore `find` invoca un blocco per ciascun elemento dell'array invocante, passando tale elemento come parametro. Non appena il blocco restituisce un valore diverso da `false` o da `nil`, l'iteratore restituisce il valore dell'elemento passato come parametro

```
array = [3,6,9,12,15]

value = array.find {|v| v > 9}
puts "First value greater than 9: =" + value.to_s
```

```
>12
>
```

Iteratori della classe `Array`

```
array_ref.map
```

L'iteratore `map` (o `collect`) invoca un blocco per ciascun elemento dell'array invocante, passando tale elemento come parametro. Per ogni elemento il blocco deve restituire un valore che `map` inserisce in un array da restituire al termine delle iterazioni

Iteratori della classe `Array`

```
array_ref.map
```

L'iteratore `map` (o `collect`) invoca un blocco per ciascun elemento dell'array invocante, passando tale elemento come parametro. Per ogni elemento il blocco deve restituire un valore che `map` inserisce in un array da restituire al termine delle iterazioni

```
array = [3,6,9,12,15]

map_array = array.map {|v| v*v}
puts map_array
```

Iteratori della classe `Array`

```
array_ref.map
```

L'iteratore `map` (o `collect`) invoca un blocco per ciascun elemento dell'array invocante, passando tale elemento come parametro. Per ogni elemento il blocco deve restituire un valore che `map` inserisce in un array da restituire al termine delle iterazioni

```
array = [3,6,9,12,15]

map_array = array.map {|v| v*v}
puts map_array
```

```
>9
>36
>81
>144
>225
>
```

Iteratori della classe `Hash`

```
hash_ref.each
```

L'iteratore `each` invoca il blocco per ciascuna delle coppie chiave-valore presenti nell'hash invocante, passando ogni coppia come parametri

Iteratori della classe `Hash`

```
hash_ref.each
```

L'iteratore `each` invoca il blocco per ciascuna delle coppie chiave-valore presenti nell'hash invocante, passando ogni coppia come parametri

```
hash = {"uno" => 1, "due" => 2, "tre" => 3}

hash.each {|k, v| puts k.to_s + " " + v.to_s}
```

Iteratori della classe `Hash`

```
hash_ref.each
```

L'iteratore `each` invoca il blocco per ciascuna delle coppie chiave-valore presenti nell'hash invocante, passando ogni coppia come parametri

```
hash = {"uno" => 1, "due" => 2, "tre" => 3}

hash.each {|k, v| puts k.to_s + " " + v.to_s}
```

```
>uno 1
>due 2
>tre 3
>
```

Iteratori della classe `Hash`

```
hash_ref.each_key
```

L'iteratore `each_key` invoca il blocco per ciascuna delle chiavi presenti nell'hash invocante, passando ogni chiave come parametro

Iteratori della classe `Hash`

```
hash_ref.each_key
```

L'iteratore `each_key` invoca il blocco per ciascuna delle chiavi presenti nell'hash invocante, passando ogni chiave come parametro

```
hash = {"uno" => 1, "due" => 2, "tre" => 3}

hash.each_key {|k| puts k}
```

Iteratori della classe `Hash`

```
hash_ref.each_key
```

L'iteratore `each_key` invoca il blocco per ciascuna delle chiavi presenti nell'hash invocante, passando ogni chiave come parametro

```
hash = {"uno" => 1, "due" => 2, "tre" => 3}

hash.each_key {|k| puts k}
```

```
>uno
>due
>tre
>
```

Iteratori della classe `Hash`

```
hash_ref.each_value
```

L'iteratore `each_value` invoca il blocco per ciascuno dei valori presenti nell'hash invocante, passando ogni valore come parametro

Iteratori della classe `Hash`

```
hash_ref.each_value
```

L'iteratore `each_value` invoca il blocco per ciascuno dei valori presenti nell'hash invocante, passando ogni valore come parametro

```
hash = {"uno" => 1, "due" => 2, "tre" => 3}

hash.each_value {|v| puts v}
```

Iteratori della classe `Hash`

```
hash_ref.each_value
```

L'iteratore `each_value` invoca il blocco per ciascuno dei valori presenti nell'hash invocante, passando ogni valore come parametro

```
hash = {"uno" => 1, "due" => 2, "tre" => 3}

hash.each_value {|v| puts v}
```

```
>1
>2
>3
>
```

Iteratori dei numeri interi

```
num.times
```

L'iteratore `times` invoca il blocco per un numero di volte pari all'intero invocante *num*. Ad ogni iterazione i (per i=1,2,…,*num*) viene passato come parametro il valore (i-1)

Iteratori dei numeri interi

```
num.times
```

L'iteratore `times` invoca il blocco per un numero di volte pari all'intero invocante *num*. Ad ogni iterazione i (per i=1,2,…,*num*) viene passato come parametro il valore (i-1)

```
7.times {|v| puts v}
```

Iteratori dei numeri interi

```
num.times
```

L'iteratore `times` invoca il blocco per un numero di volte pari all'intero invocante *num*. Ad ogni iterazione i (per i=1,2,…,*num*) viene passato come parametro il valore (i-1)

```
7.times {|v| puts v}
```

```
>0
>1
>2
>3
>4
>5
>6
>
```

Iteratori dei numeri interi

```
num1.upto(num2)
```

L'iteratore `upto` invoca il blocco per *i* che va da *num1* a *num2* (maggiore di *num1)*. Ad ogni iterazione i viene passato come parametro il valore i

Iteratori dei numeri interi

```
num1.upto(num2)
```

L'iteratore `upto` invoca il blocco per *i* che va da *num1* a *num2* (maggiore di *num1)*. Ad ogni iterazione i viene passato come parametro il valore i

```
15.upto(20) {|v| puts v}
```

Iteratori dei numeri interi

```
num1.upto(num2)
```

L'iteratore `upto` invoca il blocco per *i* che va da *num1* a *num2* (maggiore di *num1)*. Ad ogni iterazione i viene passato come parametro il valore i

```
15.upto(20) {|v| puts v}
```

```
>15
>16
>17
>18
>19
>20
>
```

Iteratori dei numeri interi

```
num1.downto(num2)
```

L'iteratore `downto` invoca il blocco per *i* che va da *num1* a *num2* (minore di *num1)*. Ad ogni iterazione i viene passato come parametro il valore i

Iteratori dei numeri interi

```
num1.downto(num2)
```

L'iteratore `downto` invoca il blocco per *i* che va da *num1* a *num2* (minore di *num1)*. Ad ogni iterazione i viene passato come parametro il valore i

```
20.downto(15) {|v| puts v}
```

Iteratori dei numeri interi

```
num1.downto(num2)
```

L'iteratore `downto` invoca il blocco per *i* che va da *num1* a *num2* (minore di *num1)*. Ad ogni iterazione i viene passato come parametro il valore i

```
20.downto(15) {|v| puts v}
```

```
>20
>19
>18
>17
>16
>15
>
```

Strutture di controllo

- Strutture di selezione:
  - `if`
  - `unless`
- Strutture di iterazione:
  - `while`
  - `until`

Selezione `if`

```
if condition1 then

  instructions1

elsif condition2

  instructions2

  …
elsif conditionN

  instructionsN

else

  else_instructions

end
```

Selezione `if`

```
if condition1 then

  instructions1

elsif condition2

  instructions2

  …
elsif conditionN

  instructionsN

else

  else_instructions

end
```

condition1, …, *conditionN* sono espressioni che restituiscono `true` oppure `false` ➔ Vengono eseguite le istruzioni relative alla prima espressione che restituisce `true`.

Selezione `if`

```
if condition1 then
  instructions1
elsif condition2
  instructions2
  …
elsif conditionN
  instructionsN
else
  else_instructions
end
```

Un `if` è un'espressione che restituisce il valore dell'ultima espressione eseguita

Selezione `if`

```
cognome = “Manzoni”

if cognome == “Manzoni” then
  puts “Alessandro”
end
```

```
>Alessandro
>
```

Selezione `if`

```
cognome = “Manzoni”

if cognome == “Manzoni” then
  nome = “Alessandro”
elsif cognome == “Leopardi”
  nome = “Giacomo”
elsif cognome == “Alighieri”
  nome = “Dante”
else
  nome = “sconosciuto”
end

puts nome
```

```
>Alessandro
>
```

Selezione `if` (come espressione)

```
cognome = “Manzoni”

nome = if cognome == “Manzoni” then
  “Alessandro”
elsif cognome == “Leopardi”
  “Giacomo”
elsif cognome == “Alighieri”
  “Dante”
else
  “sconosciuto”
end

puts nome
```

```
>Alessandro
>
```

Selezione `if` (come espressione)

```
cognome = “Manzoni”

nome = if cognome == “Manzoni” then
  “Alessandro”
elsif cognome == “Leopardi”
  “Giacomo”
elsif cognome == “Alighieri”
  “Dante”
else
  “sconosciuto”
end

puts nome
```

NOTA BENE: le variabili usate all’interno di un struttura di controllo sono disponibili anche dopo di essa

```
>Alessandro
>
```

Selezione `unless`

```
unless condition then

  instructions

end
```

Selezione `unless`

```
unless condition then

  instructions

end
```

Le istruzioni vengono eseguite a meno che la condizione non risulti vera ➔ negazione di `if`

Selezione `unless`

```
unless condition then

  instructions

end
```

```
cognome = “Leopardi”

unless cognome == “Manzoni” then
  puts cognome
end
```

Selezione `unless`

```
unless condition then

  instructions

end
```

```
cognome = “Leopardi”

unless cognome == “Manzoni” then
  puts cognome
end
```

```
>Leopardi
>
```

Iterazione `while`

```
while condition

  instructions

end
```

Iterazione `while`

```
while condition

  instructions

end
```

```
i = 0
str = “”
while i < 10
  str = str + i.to_s + “_”
  i = i+1
end
puts str
```

```
>0_1_2_3_4_5_6_7_8_9_
>
```

Iterazione `until`

```
until condition

  instructions

end
```

Negazione di `while` ➔ le iterazioni proseguono finché la condizione risulta falsa (non appena diventa vera il ciclo si blocca)

Iterazione `until`

```
until condition

  instructions

end
```

Negazione di `while` ➔ le iterazioni proseguono finché la condizione risulta falsa (non appena diventa vera il ciclo si blocca)

```
i = 0
str = ""
until i == 10
  str = str + i.to_s + "_"
  i = i+1
end
puts str
```

```
>0_1_2_3_4_5_6_7_8_9_
>
```

Gli operatori

- Operatori aritmetici
- Operatori di confronto
- Operatori logici

Operatori aritmetici

- + operatore di somma e + unario.
- – operatore di sottrazione e - unario.
- * operatore di moltiplicazione
- / operatore di divisione (se entrambi gli operandi sono interi, allora la divisione è intera)
- % operatore di modulo
- ** operatore di elevamento a potenza

Operatori di confronto

- == restituisce true se i due operandi hanno lo stesso valore.
- <, <=, >, >= sono gli operatori di “minore”, “minore o uguale”, maggiore”, e “maggiore o uguale”
- <=> restituisce -1 se il primo operando è minore del secondo, 0 se sono uguali, e +1 se il primo operando è maggiore del secondo
- `eql?` restituisce `true` se l’oggetto invocante e l’oggetto passato come argomento hanno lo stesso valore e sono dello stesso tipo
- `equal?` restituisce `true` se l’oggetto invocante e l’oggetto passato come argomento hanno lo stesso riferimento

Operatori di confronto

- == restituisce true se i due operandi hanno lo stesso valore.
- <, <=, >, >= sono gli operatori di “minore”, “minore o uguale”, maggiore”, e “maggiore o uguale”
- <=> restituisce -1 se il primo operando è minore del secondo, 0 se sono uguali, e +1 se il primo operando è maggiore del secondo
- `eql?` restituisce `true`
come argomento hann
- `equal?` restituisce `tru`
passato come argomer

Attenzione! Tutti gli operatori in Ruby sono metodi degli oggetti. Ad esempio scrivere `45 == 36` significa che l’intero `45` invoca il metodo che ha nome == passandogli come argomento l’intero `36`. E come ogni metodo anche gli operatori possono essere ridefiniti.

Operatori logici

Attenzione! In Ruby tutto è considerato vero tranne `false` e `nil`

Quindi ad esempio:

- la stringa `"gatto"`
- la stringa vuota `""`
- l'intero `45`
- l'intero 0
- il decimale `1.2`
- il symbol `:north`

sono considerati veri.

Operatori logici

Operatori di congiunzione logica ➔ **`and`** e **`&&`**

Entrambi restituiscono il primo operando falso, oppure il secondo se entrambi gli operandi sono veri.

Differiscono per il fatto che `and` ha una precedenza inferiore a quella di `&&`

Operatori logici

Operatori di congiunzione logica ➔ **`and`** e **`&&`**

Entrambi restituiscono il primo operando falso, oppure il secondo se entrambi gli operandi sono veri.

Differiscono per il fatto che `and` ha una precedenza inferiore a quella di `&&`

```
puts 0 && “gatto”
puts 45 && false
puts false && 45
puts nil && false
```

Operatori logici

Operatori di congiunzione logica ➔ **`and`** e **`&&`**

Entrambi restituiscono il primo operando falso, oppure il secondo se entrambi gli operandi sono veri.

Differiscono per il fatto che `and` ha una precedenza inferiore a quella di `&&`

```
puts 0 && “gatto”
puts 45 && false
puts false && 45
puts nil && false
```

```
>gatto
>false
>false
>nil
>
```

Operatori logici

Operatori di congiunzione logica ➔ **`and`** e **`&&`**

Entrambi restituiscono il primo operando falso, oppure il secondo se entrambi gli operandi sono veri.

Differiscono per il fatto che `and` ha una precedenza inferiore a quella di `&&`

```
puts 0 && “gatto”
puts 45 && false
puts false && 45
puts nil && false
```

`and` e `&&` restituiscono quindi un valore vero solo se entrambi gli operandi sono veri.

```
>gatto
>false
>false
>nil
>
```

Operatori logici

Operatori di disgiunzione logica ➔ **`or`** e `||`

Entrambi restituiscono il primo operando vero, oppure il secondo se entrambi gli operandi sono falsi.

Differiscono per il fatto che `or` ha una precedenza inferiore a quella di `||`

Operatori logici

Operatori di disgiunzione logica ➔ **`or`** e **`||`**

Entrambi restituiscono il primo operando vero, oppure il secondo se entrambi gli operandi sono falsi.

Differiscono per il fatto che `or` ha una precedenza inferiore a quella di `||`

```
puts 0 || “gatto”
puts 45 || false
puts false || 45
puts nil || false
```

Operatori logici

Operatori di disgiunzione logica ➔ **`or`** e **`||`**

Entrambi restituiscono il primo operando vero, oppure il secondo se entrambi gli operandi sono falsi.

Differiscono per il fatto che `or` ha una precedenza inferiore a quella di `||`

```
puts 0 || “gatto”
puts 45 || false
puts false || 45
puts nil || false
```

```
>0
>45
>45
>false
>
```

Operatori logici

Operatori di disgiunzione logica ➔ **`or`** e **`||`**

Entrambi restituiscono il primo operando vero, oppure il secondo se entrambi gli operandi sono falsi.

Differiscono per il fatto che `or` ha una precedenza inferiore a quella di `||`

```
puts 0 || “gatto”
puts 45 || false
puts false || 45
puts nil || false
```

`or` e `||` restituiscono quindi un valore vero solo se almeno un operando è vero.

```
>0
>45
>45
>false
>
```

Operatori logici

Operatori di negazione logica ➔ **`not`** e **`!`**

Entrambi restituiscono vero se l'operando è falso, e falso se l'operando è vero.

Differiscono per il fatto che `not` ha una precedenza inferiore a quella di **`!`**

Operatori logici

Operatori di negazione logica ➔ **`not`** e **`!`**

Entrambi restituiscono vero se l'operando è falso, e falso se l'operando è vero.

Differiscono per il fatto che `not` ha una precedenza inferiore a quella di `!`

```
puts !0
puts !45
puts !false
puts !nil
```

Operatori logici

Operatori di negazione logica ➔ **`not`** e **`!`**

Entrambi restituiscono vero se l'operando è falso, e falso se l'operando è vero.

Differiscono per il fatto che `not` ha una precedenza inferiore a quella di `!`

```
puts !0
puts !45
puts !false
puts !nil
```

```
>false
>false
>true
>true
>
```